A REVIEW

of

NON-ADA to ADA CONVERSION

August 5, 1993

Prepared for:
Rome Reasearch Site
Rome, NY 13441

Prepared by:
ITT Industries System & Sciences Corporation
775 Daedalian Drive
Rome, New York 13441-4909

APPENDICES

FORWARD

This report, *A Review of Non-Ada to Ada Conversion*, provides a discussion of the processes and problems involved with the conversion of software from early High Order Languages (HOLs) to Ada. Specific compatibility and format difficulties are discussed for each HOL examined, including C, Cobol, Fortran, Jovial, Modula-2, and Pascal. Several conversion approaches are presented, along with the advantages and disadvantages of each. Three specific conversion projects are overviewed. Several currently available Off the Shelf (OTS) tools to help the conversion process are included for reference, with a short discussion of each.

1. INTRODUCTION

After their original design and implementation, many software systems have been redesigned and converted from one platform to another. Each time the decision has to be made, long agonizing hours are spent to determine the approach to take for the conversion. Early language conversions were from one assembly language to another as hardware evolved past the current system. Later conversions were from assembly language to a High Order Language (HOL), or from one HOL to another, as application-oriented HOLs developed. The introduction of the military-standard programming language Ada and its mandated use on Department of Defense (DoD) systems has created a need to convert existing systems from other HOLs to Ada. It is with the conversion to Ada that this report is concerned.

1.2 Ada and the Need for Conversion

Ada was designed and developed to fill a need for better control and management of DoD projects, especially projects concerning embedded systems. An embedded system is a system incorporated into a larger system such as ships, airplanes, satellites, and command and control systems (Shumate 84). Embedded systems are usually environment-driven, interfacing with external interrupts which service real-time operations. These systems can be over 100,000 source lines of code, are used over a long period of time, and are frequently modified and enhanced. Since embedded systems often involve human lives or mission-critical functions, reliability and maintainability are prime concerns.

Since the early 1970s, the DoD has spent an increasing proportion of their software budget on embedded systems and the maintenance of existing systems. Studies revealed that DoD systems operated on over 200 computer models and were written in over 450 programming languages and dialects. Many of these systems used a large proportion of assembly language to meet requirements associated with embedded systems that could not be satisfied by HOLs. These characteristics of DoD systems created a number of problems associated with cost and maintainability. The cost of developing compilers and other software tools for many different languages with few

common features was excessive. Many projects used new unproven tools with consequent reliability and training problems. Maintenance by other than the original developers was difficult since only a small number of organizations had any competence in many of the languages. Finally, programmers needed continuous training and retraining (Shumate 84).

The DoD brought the High-Order Language Working Group (HOLWG) together in 1975 to formulate requirements for a solution to these problems, and to begin with the evaluation of existing languages. A Policy framework was established by DoD Directive 5000.29, *Management of Computer Resources in Major Defense Systems*. The directive required, for all new software development, the use of a high-order HOL selected from a list of approved languages: ANSI Cobol, ANSI Fortran, CMS-2, J3 Jovial, J73 Jovial, SPL/I, and Tacpol. To formulate the capabilities a new DoD language would have to fulfill, the HOLWG then developed a series of evolving requirements documents identified as Strawman, Woodenman, Tinman, Ironman, and Steelman. Each successive document was distributed to the interested community for comments, additions and modifications. The final version, Steelman, defined criteria for the design of a new HOL.

Pascal was chosen as the basis of the new Ada Language, and in June 1979 the Preliminary Ada Reference Manual was distributed to over 10,000 individuals for review. The Ada design was formally accepted by the HOLWG on August 25, 1980, and was adopted as a military standard, MIL-STD-1815, on 10 December 1980. The *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, was adopted as an American National Standards Institute (ANSI) standard in February 1983 and is still the defining document for Ada.

The DoD has mandated that Ada be used as the development language for all new mission-critical software. DoD Directive 3402.1 dictates when Ada should or must be used for development, when it should or must be used for conversions, and under what conditions exceptions to the mandate are accepted. Along with this mandate, each branch of the government and military issue their own mandates which enhance the DoD mandate for their own purposes. In general all new software development must be done in Ada, and when a hardware/software upgrade of an existing system

requires modification of more than a third of its code over the system life cycle the transition to Ada is required.

The introduction of Ada and the DoD mandate required many programmers and systems to be transitioned to Ada. A body of experience has been accumulated on how to convert existing systems to Ada. This report summarizes that experience.

1.3 Report Overview

The details of how to perform a conversion differ from system to system depending on the source and target HOLs. Nevertheless, similarities exist which can be generalized. From these generalizations come conversion techniques and tools which can shorten the conversion process. The basis for these conversions aids is overviewed in this report.

Section 1 consists of this introduction which describes why Ada was introduced, the need to convert existing systems to Ada, and the remainder of the report.

Section 2 summarizes important features and characteristics of Ada and different approaches for converting a system to Ada.

Section 3 presents a survey of the state of the art for converting systems to Ada. This survey is decomposed based on the source HOL. C, Fortran, Jovial, Pascal, Modula-2, and Cobol are all considered. Three concrete case studies are summarized. One of these is an internal Kaman Sciences project that converted a system written in C to Ada.

Section 4 presents certain conversion aids and tools.

Section 5 summarizes the report findings. References are provided in Section 6. Appendices contain Ada source code for utilities that were developed during the internal Kaman Sciences project described in Section 3.

2. THE CONVERSION PROCESS

A discussion of the conversion process with the intent of providing direction to managers who are about to attempt such a conversion is provided in this section. Since an Ada system is the desired end product of the conversion process, relevant characteristics of Ada are first described. Second, various conversion approaches are discussed at the general level. Third, suggestions on how to approach a conversion project are offered.

2.1 Characteristics of Ada

Ada is a programming language designed in accordance with the Steelman requirements, defined by the United States Department of Defense. Overall, these requirements call for a language with considerable expressive power covering a wide application domain. As a result, the language includes facilities offered by classical languages such as Pascal, as well as facilities often found only in specialized languages. Thus the language is a modern algorithmic language with the usual control structures, and with the ability to define types and subprograms. It also serves the need for modularity whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation. In addition to these aspects, the language covers real-time programming, with facilities to model parallel tasks and to handle exceptions. It also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, both application-level and machine-level input-output are defined (RM 83).

Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. Emphasis was placed on program readability over ease of writing, and an attempt was made to cover the domain with a small number of underlying concepts integrated in a consistent and systematic way. Modern software engineering principles were embodied in the language, especially with respect to methods related to "programming in the large." In other words, Ada and its environment supports the development of large systems by teams of programmers. Ada can also be used as a design language.

The Ada *package* is the main feature used to structure Ada systems at the highest level. A package provides its users with a collection of constants, data types, variables, and subprograms. Packages allow designers to develop a system as a collection of modules, where each module encapsulates tightly coupled elements. Ada packages come in two parts, a specification and an implementation. The specification defines the interface to the outside world, while the implementation shows the details of how a package is implemented. Through the use of *private types*, the implementation of a data type can be separated in the private part of a package specification from its declaration. For example, the implementation of a stack as an array or linked list can be separated in a specification from its declaration. Only the declaration will be visible to users of the package. These features of the Ada programming language support the software engineering concepts of modularity, information hiding, and abstract data types (Stock).

Ada is a strongly typed language, supporting the requirement of reliability. In a well-designed Ada program, programmers cannot make mistakes such as adding a real number to a program address. Extensive type structuring mechanisms, including subtypes, derived types, arrays, records, and variant records, provide a rich set of features for creating user-defined types out of the simple predefined types. The programmer can create a collection of types that mirror the objects in the application domain.

In most other strongly typed languages programmers cannot easily create families of data structures. For example, a stack of characters and a stack of integers are implemented by different types. This limits the ability of reusing algorithmic code that is independent of the type of the objects to which it applies. Furthermore, it is difficult to create general reusable libraries. Ada does not suffer from these limitations; Ada *generics* allow programmers to overcome the limitations of strong typing in a controlled fashion. For example, an Ada generic package can be written for stacks that is independent of the type of objects stored in the stack. Different *instantiations* of the generic package can be used to create different stack types. Commercially available Ada libraries of reusable components, including standard data structures, rely heavily on generics.

Ada was developed to specifically support embedded systems. These systems have high reliability requirements that differ from traditional systems. For example, the absence of a data file should not cause a flight control system to generate an error

message and then crash. Embedded systems should be robust in these exceptional situations. Ada provides *exceptions* to handle just these conditions. Exceptions, which can be user-defined, are raised when unforeseen occurrences arise. Examples include division by zero, an inability to dynamically allocate memory when needed, and an attempt to index a nonexistent array element. Exception handlers allow system designers to create software in which processing is resumed in a controlled fashion at the appropriate level.

Embedded systems may also involve multitasking and parallel processing. Ada *tasks* and *task types* are built-in language features for this purpose. Ada provides an extensive tasking model which allows system developers to ensure tasks properly rendezvous. If required, one task can be suspended until another is required, or the first task can continue by proper use of the Ada *select* statement. Proper design can protect a critical resource from being simultaneously used by several tasks. An Ada design can also enforce different priorities for different tasks.

Another important feature of Ada is the ability to control system level details and a program's relation to its environment. *Pragmas* are compiler directives which allow the programmer to alter the order in which packages are compiled, to control the dynamic allocation and deallocation of memory, to interface Ada programs to object files generated from code written in other languages, to control type checking, to control optimization, etc. As a further feature for allowing programmers direct access to system level features, *representation clauses* permit the specification of how types map onto the underlying machine. They are useful in accessing unusual peripheral hardware devices, for controlling storage requirements, for ensuring memory alignments for certain types, for address arithmetic, for processing interrupts, for interfacing with machine language, and for other functions often performed in assembly language. These features are often provided by implementations in other languages, but they have rarely been included in language definition standards before.

As can be seen from this brief overview, Ada provides a rich set of features. The proper use of these features requires a software engineering discipline and results in a program written in a certain style. The Ada language contains nearly every general capability and most special capabilities of existing HOLs. Because of this property, one method of converting a non-Ada system to Ada relies on determining how each of the source language capabilities can be mapped into Ada. Most problems which arise

involve data type formats, parameter passing and common data sharing. These problems are defined and discussed for each of the HOLs covered in Section 3. The result of conversions which rely on approaches requiring a one-to-one mapping from the source language to Ada will not take full advantage of Ada features and will result in an Ada program written more in the style of the source language than typical Ada.

Converting non-Ada to Ada without modifying the system to make use of the Ada capabilities is still a worth-while endeavor. The resulting program will take advantage of Ada software's portability, maintainability, extensibility and self-documenting features. It will also gain the benefit of the extensive Ada infrastructure within the DoD and their associated contractors. Tools, libraries of reusable components, and a well-trained workforce will be available for the converted program. Future enhancements can be written in a more appropriate Ada style, and the system can gradually evolve to a well-designed Ada program.

2.2 Conversion Approaches

Depending upon the resources available, several approaches to the conversion process may be taken. They differ in the time required and how close the final result will be to a design originally intended for Ada. These approaches are discussed in the following subsections along with the advantages and disadvantages of each:

- o Complete redesign and rewrite
- o Incremental functionally equivalent replacement
- o Incremental redesign and rewrite
- o Incremental multilingual rewrite
- o Automatic translation

2.2.1 Complete Redesign and Rewrite

The most thorough method of converting a system to Ada, and the most sure method of ensuring that the features of the Ada language are fully and properly exploited, is to completely redesign and rewrite the system. This approach involves a complete analysis, design, implementation, and test of the system, thereby making the best use of Ada's software engineering capabilities.

The cost and duration of such a conversion will most likely be comparable to that of the original development. If the organization performing the conversion is undertaking this project as part of a transition to Ada, they will most likely have a small supply of experienced Ada developers and programmers. Including training costs, the total cost of the conversion will probably exceed that of the original development. All "experience learned" from this approach, however, will be utilized in the maintenance and enhancement of the system during its lifecycle. Consequently, the total lifecycle cost may decrease.

This approach is best for badly written original source code or code which has evolved beyond the original good design and is difficult to maintain and modify in its original language. It is generally recognized that bad source code going into a conversion of any kind will result in bad Ada code. It may also be an appropriate approach if constraints of the original language resulted in a much worse design than would be the case in Ada. Thus, the resulting Ada system can be expected to have a much more elegant design with consequent improvements in maintainability. In all cases, the costs of this approach must be balanced against the expected decrease in operations and maintenance costs.

2.2.2 Incremental Functionally Equivalent Replacement

Another conversion approach is to incrementally replace modules with functionally equivalent Ada code. This approach is appropriate when the original design is still valid and the source language code was well designed, developed, and documented using good software engineering principles. The intent is to use the "black box" method to replace each module of the original system with an Ada subprogram (procedure or function) which requires the same input/output parameters (interface) and exhibits the same functionally as does the original module. Every attempt is made to use Ada capabilities within the "black box".

The final system resulting from this approach would be structured around Ada subprograms, not Ada packages. If the conversion proceeds in a strictly top-down fashion, the Ada code created during intermediate steps will need to call procedures and functions written in the original language. If a bottom-up approach is adopted, portions of the original system will call Ada subprograms during intermediate steps. In any case, interfaces between the original language and Ada will continually need to be

implemented. The location of these interfaces changes as the process continues, with more and more of the system gradually being transformed to Ada. At completion the entire system is then in Ada.

An advantage to this approach is that it can be performed while the system is still being used under production. Functionally equivalent modules are unit tested and then placed into the system. If the interfaces are designed and implemented correctly there should be no impact upon the production system with the exception, hopefully, of an increase of efficiency and stability supplied by Ada.

A disadvantage to this approach is that the converted modules must contain knowledge of the unconverted calling modules due to the differences in data types and formats which are shared as global data and passed as parameters. Data types and formats are discussed more fully in Section 2.2.4 below. If the conversion is from Fortran to Ada, for example, the burden of Fortran-to-Ada and Ada-to-Fortran data type conversion falls upon the Ada module so as not to disturb the original calling modules. This approach results in an Ada system in which its modules interface using non-Ada data types. This problem can be handled in two ways:

(1) By creating two overloaded modules for each replacement, one with a Fortran interface and one with the target Ada interface

(2) By a cleanup pass at the end of the conversion to modify the modules with the target Ada interfaces.

The first method allows initial design and implementation of Ada concepts within the Ada interface overloaded module such as exceptions and generics, but requires duplicate code which must be kept in sync as development progresses. This method, however, allows newly converted modules to use the Ada interface of modules already converted. Appropriate scheduling of module conversion should be able to take advantage of this to lower the number of required overloaded modules. The second method does not require the module duplication for dual interfaces, but introduces a system-wide 'interface test' at the end which would most likely be more extensive than the module duplication of the first method.

Another disadvantage to this approach involves desired Ada capabilities. The resulting system makes little use of the software engineering capabilities of Ada. No structuring mechanism above the subprogram level is used. In particular, Ada packages, with their attendant modularity, data abstraction, and information hiding capabilities are not used. Furthermore, unless the "main" program is first converted to Ada and the Ada runtime system is used, Ada exceptions cannot be introduced during this approach.

A method to take the sting out on this approach would be to collect groups of modules into "packages", and to place a non-Ada interface around each package. In this way, Ada interfaces could be used between the internal modules of each package. This would require final manipulation and testing of the package interfaces rather than module interfaces. This method, however, requires package testing and replacement of the included modules into the production system as a group.

2.2.3 Incremental Redesign and Rewrite

In a large system with a set of intercommunicating subsystems, the conversion can be performed incrementally one subsystem at a time. This is similar to the "Incremental Functionally Equivalent Replacement" approach described in Section 2.2.2, except a subsystem at a time is rewritten rather than just a single module or subprogram. The approach still requires well defined interfaces such that the remaining unconverted subsystems have no knowledge that converted subsystems are now in Ada rather than in the source language. The converted subsystems will be completely redesigned and developed to make full use of the software engineering capabilities of Ada.

This approach requires subsystem testing and replacement of the source subsystem interfaces with Ada subsystem interfaces. These tests will be conducted much the same as with the "Incremental Functionally Equivalent Replacement" approach when source modules are grouped together to be replaced by Ada packages.

An advantage of this approach is that conversion of the entire system need not be undertaken at the same time. Subsystems which are not slated to be maintained much longer need not be converted, yet they can still be used along with the converted subsystems until they are phased out.

2.2.4 Incremental Multilingual Rewrite

The result of an incremental multilingual rewrite is a system that is only partially in Ada. A part remains in the original source language. This approach would be used when only a portion of a system, such as a user specific library, is to be converted to Ada while an Off-the-Shelf (OTS) package, such as a User Interface or Database Interface library, will remain in a non-Ada language and will be linked with the new Ada code.

A determination must be made on which language dominates, thus determining which run-time system will be in control. This decision is usually determined by the language in which the "main" program is written, and which loader is used to link the executable file. Often when using an OTS user interface package a main program is supplied with the package which interacts with the workstation or window software. If this is the case, there are two plans of attack.

First, the original source language run-time system can be used, preserving the main routine in that language. This plan, however, precludes the use of several of Ada's more valuable capabilities; exceptions, elaboration and instantiation. Although one can make do without exceptions, trouble occurs if any pre-existing Ada libraries are used in which exceptions are already programmed. The non-Ada run-time system generally cannot handle exceptions passed to it. Data elaborations can be replaced with the standard initialization code at the start of each module. The loss of generic instantiations, however will be felt when trying to introduce Ada capabilities into the conversion.

The second plan requires the creation of a dummy Ada main program which then calls the original non-Ada main program as its only subroutine. In this way the Ada run-time system is in charge and all of the capabilities of Ada libraries can be used. The non-Ada modules are treated properly by the Ada PRAGMA mechanism. This is by far the better plan. It will take a little work and some digging in the system manuals to find out how to accept the command-line arguments in the Ada main program and then to pass them to the non-Ada main program so that it thinks it is still in charge. An example of this can be found in the source code presented in the appendices to the

report. These were generated under the Kaman's Internal Research and Development Project "C-to-Ada Conversion," described in Section 3.2.3. This plan also offers an easy method for module replacement. The Ada-Linker command string will be added to the non-Ada module library. The new Ada main program will start with a set of Ada WITH statements which name the packages containing the newly converted modules. Since the Ada linker searches the Ada library before any foreign library(s), only the converted modules which have been "WITHed" in the main routine will be used. Non-WITHed modules will be drawn from the non-Ada module library(s).

Once the appropriate run-time system has been chosen, the next step is to attack the problem of parallel data types. This is the process of creating Ada data types which mimic those of the non-Ada language with which the converted Ada library will be linked. Parallel data types are used for the input/output parameters to and from the non-Ada modules, and constitute the non-Ada interface discussed in Section 2.2.2 above. Parallel data types are also used to access global data shared between the non-Ada and Ada modules. Global data is accessed via the INTERFACE_NAME pragma, while the non-Ada modules are accessed via the INTERFACE and INTERFACE_NAME pragmas. Appendix A is an example of the parallel data types and module declarations for a C to Ada multilingual interface. The non-Ada modules in this case are a subset of UNIX system routines. Although the INTERFACE pragma is defined by the RM (RM 83) and therefore supported by all Ada compilers, the INTERFACE_NAME pragma is not. If the target Ada compiler does not support the INTERFACE_NAME pragma, another method of global data sharing between the non-Ada and Ada modules must be derived.

2.2.5 Automatic Translation

Automatic translation uses automatic and/or semi-automatic tools for a direct translation from a source HOL to Ada (Parsian 88). This approach yields a similar result as the approach discussed in Section 2.2.2, but has many problems in areas such as tasking, Operating System calls, assembly language inserts, parallel data types, etc. Although the original comments and documentation are usually preserved by the tools, the resultant code is often difficult to read and understand. This leads to maintenance and enhancement difficulties.

Generally the Automatic Translation approach involves a step to convert the source code into some intermediate format, and then a step to convert the intermediate format into the target (Ada) source code. For each source language, therefore, there will be a conversion mechanism to put it into the intermediate format, and there will be a conversion mechanism to put the intermediate format into each target language. This offers considerable flexibility for the addition of new source and target languages to the tool. Some tools do no more than a straight conversion in and out, while others offer restructuring of the source language code to remove some of the source language problems and to prepare for the output conversion to a structured or object oriented language. Such a case would be the removal of GOTOs from Fortran and Cobol.

Several OTS tools are discussed later in this report. Each of the translation tools has its own advantages and disadvantages and may offer anywhere from partial translation, requiring human intervention during or after the conversion process, to 100% translation. Generally these tools create a functionally equivalent copy of the original system, using only those Ada capabilities which were available in the original language model (Horton 85).

Translation tools can, of course, be used to translate individual modules or groups of modules to facilitate any of the above discussed approaches. This use of the translation tools would tend to conserve resources, even if the translated modules were then enhanced to use Ada capabilities before entrance into the system. In multilingual conversions, however, this usefulness is limited by the extent to which the tool can devise and manipulate parallel data types and shared data.

The technique of parallel data types extends to total translation in that not only the language's built in data types, but any user defined data types as well must be converted to Ada data types throughout the system. The difficulties of this conversion process for well-known HOLs are discussed more in Section 3.1.

Stock comments that with Automatic Translation one cannot take nearly as much advantage of the similarities of the source and target language, especially if these similarities are not shared with the intermediate language. This is a particular problem with Pascal, as Pascal and Ada are quite similar (Stock).

2.3 Approach Suggestions

Any conversion effort has a number of managerial implications and raises certain technical considerations. This section discusses a number of issues that arise with Ada conversions, no matter which of the above approaches is adopted.

2.3.1 Managerial Implications

The physical translation of a software system from a non-Ada language into Ada is only one portion of the conversion effort. Unless managerial changes are made, the effort will suffer many problems and set backs. Wallis (85) contends that "A change of language implies a change in ways of thought suggested by the constructs of the language; transition to Ada makes no sense unless there is a major retraining effort to make sure that the capabilities of the language will be properly used." Although this implies that the "mind-set" of the programmer is important, the "mind-set" of management, from task manager up to top management, must also be modified to understand and adhere to the new Software Engineering concepts required to make full use of Ada.

Concepts such as Reusable Software, Object Oriented Design (OOD), and Object Oriented Programming (OOP) must be given attention during training at the onset of the project. To expect the development team to learn all of this during execution of the project is the first large mistake often made by management in traditional project management. Ada, which includes a partial implementation of Object Oriented programming concepts, is not "just another programming language". Ada is a new way of looking at the problem solving process, and must be approached as such. Using an available set of CASE development tools which cover the entire life cycle can be very helpful at this point. And finally, not only must the development team be conversant with Ada, but they must be conversant with the source language as well. If team members are added because of previous Ada experience, it may be necessary to train them in the source language to avoid a time consuming knowledge-drain on the rest of the team.

Another management viewpoint often taken towards conversion projects is that since the conversion embodies the working system, then the converted system should

require minimal testing. This is not the case, no matter what conversion approach is chosen. The converted system must be put through the same amount of unit, system, and acceptance testing as was the original system during its implementation.

The last, but certainly not the least, important aspect of conversion is proper documentation. One of the advantages of Ada is its "self-documenting" features. This, however, is not enough to ensure ease of future maintenance and enhancement. Documentation must be written to the Ada "mind-set," which again differs between Ada and traditional programming languages.

2.3.2 Technical Considerations

When embarking upon a conversion effort, there are a number of technical considerations which should be clearly defined up front. A good library must be collected, which will contain all of the documentation for the system to be converted. This should include not only the deliverable documentation, but all of the internal notes, messages and thought papers which were written by the development team. Last, but not least, the library should contain all of the current source code files and examples of all of the input, intermediate and output data files and reports. Any test data and documentation used during the various test phases of the original development must also be collected. All of this should be held totally separate from the maintenance team, so that the conversion team can do what they will without impact upon the production system.

An initial team subset should then first become familiar with the collected data, and should organize a conversion approach and effort plan by performing at least the following steps:

- Locate and acquire any CASE and other support tools which can be of use during the conversion effort

- Identify subsets of the current language that will present conversion difficulties

- Identify modules and data structures which will not pass correctly through a translation function, if a translator is being used

- Identify modules and data structures which will need to be redesigned to make better use of Ada capabilities

- Determine where Ada EXCEPTIONS will be utilized and where they will not

- Identify all points where subroutines are passed as parameters (in source languages which support this)

If the conversion approach taken involves a multilingual flavor, the following steps should be followed:

- Determine which run-time library will be used, and develop the appropriate main program

- Identify and define all needed parallel data types, as well as any functions / procedures which will be needed to convert between the parallel data types and standard Ada types

- Identify modules and data structures which can be left in the source language with appropriate multilingual interfaces

- Determine if and where Ada type security is needed and where it must be subverted through the Ada SUPPRESS mechanism

Once all of the above information is gathered, the team is ready to begin.

3. SURVEY OF THE STATE OF THE ART

This survey reviews five of the more popular Higher Order Languages (HOLs) and the problems which pertain to each during conversion to Ada. Although there is still a need for conversion from assembly language to Ada, this report deals mainly with non-Ada HOLs. Also included in this survey are experiences with non-Ada to Ada conversions within both the DoD and industry.

3.1 HOL to Ada Conversion Problem Areas

Each of the early HOLs has its own problem areas when it involves conversion to Ada. Generally these problem areas include:

- o Intermodule communication
- o Data types and their formats
- o Operating system interfaces
- o Unavailability of structured constructs
- o Methods of global data access

It is not the intent of this Section to outline the requirements for or steps to follow when building conversion tools, but rather to bring to the reader's attention the most troublesome points for conversion of popular HOLs to Ada. Five languages are discussed most often in the literature on conversions to Ada: C, Cobol, Fortran, Jovial, and Pascal. In addition, this report discusses Modula-2, which closely resembles Ada.

3.1.1 C

C is a powerful language which can be used to create very good, well structured modules which follow the best software engineering rules. C can also be misused to create very bad, unstructured code which closely resembles early Fortran code. The former can be converted to Ada with much grace, the latter will probably need complete redesign and development. Although there is a "GOTO" construct within C,

very few C programmers used it. Well written, short, well designed C subroutines with one entry and one exit tend to translate neatly to Ada. Although C and Ada have completely different backgrounds, good design and coding practices yield easily converted routines.

The most commonly used C simple data types, int, long, short, char, unsigned int and unsigned short, can be converted to parallel data types in Ada. Examples of these parallel data types for the Sun Ada 1.0 compiler and toolset can be found in Appendices A and B and are discussed in Section 3.2.3. Difficulties arise with complex C data types in interfacing C and Ada programs.

When a C structure is passed from a C module to an Ada module, the items within the C structure may not line up with a corresponding Ada record structure due to word/byte/bit alignment by the C and Ada compilers. This is particularly true when the C structure contains pointers to other C structures. Although C structures using the simple data types, integer, long, short and char, can be aligned using the "AT i RANGE n..m" representation for the underlying bit assignment, non-char C pointers do not seem to carry through properly. This is particularly true in the case of pointers such as "struct first **third" where the pointer "first" is a member of a parent structure being passed to an Ada module. The double indirect pointer is difficult to accomplish correctly in Ada.

Non-char pointers, such as '"int *value", can be passed to an Ada module by using the "FOR a USE AT b" representation. In the following example the Ada module is called as "Ada_module_called_from_C(&int_parameter):"

```
function Ada_module_called_from_C(int_ptr : system.address) return c_string is
        int_parameter : array(1..1) of c_int;
    for int_parameter use at int_ptr;
return_string : string(1..128);
begin
  ...
     int_parameter(1) := int_parameter(1) + 10;  -- add 10 to callers parameter
  ...
return_string := "this is the string returned to C module" & ASCII.NUL;
return address_to_c(return_string(1)'address);
```

end Ada_module_called_from_C;

The example shows how the pointer to an integer parameter passed to a C function is handled when the receiving C function is replaced by an Ada function. Although an Ada function cannot pass modifications back through parameters, this device bypasses the Ada mechanism through the "system'address" data type. This example also illustrates how C arrays are passed to Ada modules. The parallel data types "c_string" and "c_int," as well as the conversion module "address_to_c" can be found in Appendices A and B.

C "subprogram parameters," that is pointers to subprograms, must be translated to "formal subprogram" parameters to an Ada generic module. This requires a change in the way the called subprogram is viewed, as well as the format of the subprogram calling point(s). Parameters to the formal subprogram also require detailed observation and possible modification.

CASE statements within C allow the omission of the "break" key word, which allows the execution of the statements in the next case alternative. Ada does not support a similar capability. Ada also requires the presence of the DEFAULT case definition for every CASE block.

3.1.2 Cobol

Cobol was developed over three decades to address business applications. Many new Management Information Systems that would formerly have been developed in Cobol are now developed under a commercial Off-The-Shelf relational Data Base Management System (DBMS) with various fourth generation query and report-writing languages. Nevertheless, there still is a large body of existing code written in Cobol, some of it for military applications.

Cobol systems are structured quite different than Ada systems, with many features being provided specifically for business applications. The toughest part of Cobol to Ada conversion would be the Data Division in Cobol programs, as there is nothing to compare with it in Ada. The Data Division structures would be transferable to Ada record structures, but the automatic editing and Cobol PIC capabilities would have to be handled with added user coded capabilities and user defined data types.

This is true with most of the data display and manipulations portions of a Cobol program.

Cobol provides certain high-level business application-specific features that are not built into the Ada language. Very little of the Cobol file handling features can be mapped to Ada Input/Output features. User supplied Ada modules are required to extend the Ada IO package to handle these features. The built in SORT feature of Cobol has no comparable feature in Ada and must be replaced with external sort capabilities.

The executable portion of a Cobol program is structured as a series of "paragraphs." The flow of control through these paragraphs can get quite complicated. The Cobol computed GOTO, "GO TO ... DEPENDING ON ..." can usually be replaced with an Ada CASE structure. Restructuring may well be needed in Cobol programs which make extensive use of the GOTO statement. Indeed for such a system any approach other than a complete redesign and rewrite may not be the best choice unless a tool which provides automatic restructuring, such as that discussed in Section 4.3, is used.

3.1.3 Fortran

Fortran, having been designed in the 1950s, is one of the oldest languages still in widespread use. It was not designed to support the development of modular systems that follow current software engineering rules. It was too early for that, and the programming field was young. Many early Fortran programs were converted or reprogrammed from assembly language code, by assembly language programmers who used Fortran as an assembler. The GOTO statement, equivalent to an assembly JUMP instruction, was often overused. When some structured constructs were later added, they did not go quite far enough to allow well written structured code. Statement numbers, GOTOs and COMPUTED GOTOs still hung around and will have to be restructured when converting to Ada.

Variables in Fortran are either simple data types or arrays. These can be converted directly to Ada, although a less automated approach would allow the use of higher-level Ada data structuring features to capture design intentions not expressible directly in Fortran source code. About the only feature in Fortran that allows heterogeneous data types to be grouped together, as in an Ada record, is the COMMON

statement. The Fortran COMMON statement allows the aliasing of a cell by two or more names. Even though the Ada pragma SHARED seems close to this capability, Martin (85) warns that this pragma applies to tasks only and can cause erroneous programs if used in conjunction with data. One method of handling Fortran COMMON and EQUIVALENCE statements is discussed under Section 3.2.1.

A widespread practice in Fortran is the use of function names as arguments to Fortran procedures. For example, a root finding procedure might include as an argument the name of a function whose roots are desired. These "subprogram parameters" must be translated to "formal subprogram" parameters in an Ada generic module. This requires a change in the way the called subprogram is viewed, as well as the format of the subprogram calling point(s). Parameters to the formal subprogram also require detailed observation and possible modification.

Many other problems of detail will arise when converting a Fortran program to Ada. For example, in Fortran, loop counters can be accessed outside the loop for various purposes, where within Ada, a second variable must be used to export the loop counter outside the loop. This is because the loop counter is local to the loop and has no existence outside the loop (Martin 85).

Restructuring may well be needed in very old Fortran programs which make extensive use of the GOTO statement. Indeed for such a system any approach other than a complete redesign and rewrite may not be the best choice unless a tool which provides automatic restructuring, such as that discussed in Section 4.3, is used.

3.1.4 Jovial

Ehrenfried (82) discusses in great detail the problems of automatically converting Jovial programs to Ada. Of particular interest are the problems dealing with Jovial table structures. In Jovial, Ehrenfried points out, the programmer can specify the layout of tables in memory, where Ada offers the programmer no control over the manner in which arrays of records are laid out. For table entries Jovial provides for three levels of packing, where Ada provides for only one level through the pragma PACK. These problems are both application and platform dependent, and must be attacked appropriately by the conversion team.

Another problem related to data structures and low-level features arises from a Jovial capability that allows objects or portions of objects to occupy the same storage space. This is similar to the Fortran EQUIVALENCE statement. Ehrenfried cautions that using the Ada "SYSTEM.ADDRESS" representation to overcome this problem is not correct, as stated in Section 13.5 of the RM. When considering a multilingual conversion approach, however, address clauses cannot be completely avoided, and indeed can be used to advantage.

Certain problems arise with control flow in Jovial that are similar to issues which arise in converting other languages to Ada. Like C, CASE statements within Jovial have a key word FALLTHRU which allows the execution of the statements in the next case alternative. Ada does not support a similar capability. Ada also requires the presence of the default case definition for every CASE block. As in Fortran, Jovial loop counters can be accessed outside the loop for various purposes, where within Ada a second variable must be used to export the loop counter outside the loop. Once again, this is because the loop counter is local to the loop and has no existence outside the loop.

In general, a conversion from Jovial to Ada is unlikely to create a program written in a typical Ada style. Many of the advanced features of Ada that permit the expression of design decisions in will not be used in the converted program.

3.1.5 Modula-2

Modula-2 is a descendant of Pascal, as is Ada. Modula-2 was developed by Niklaus Wirth, the designer of Pascal, and Wirth's textbook (Wirth 83) is an authoritative description of Modula-2. Modula-2 incorporates many software engineering concepts treated in Ada, the two languages having been designed during approximately during the same time period. Consequently, these two Pascal-derivatives are very similar languages. In many ways, Modula-2 resembles a streamlined Ada. Converting a system from Modula-2 to Ada is probably the easiest of all language conversions to Ada. Yet problems arise even here.

Modula-2 systems are organized around Modula-2 modules, just as Ada systems are organized around Ada packages. A module, like an Ada package, groups together related constants, data types, variables, and procedures. A module has a definition and implementation, corresponding to an Ada package specification and body. Given this

close resemblance between Modula-2 modules and Ada packages, the high-level structure of a Modula-2 system can often be converted directly to Ada by a one-to-one mapping of modules to packages.

Case is one minor property of Modula-2 that causes programs to initially appear different at a surface level. Modula-2 is case-sensitive, and all language keywords must appear in uppercase. The convention fostered through Wirth's book is that user-defined variables are entirely in lowercase, while other user-defined identifiers often appear in mixed case. Ada is not case-sensitive, so a direct conversion of modules to packages need not change the Modula-2 convention. On the other hand, the stylistic convention in the Ada community, as illustrated in the RM, is exactly the opposite. Ada keywords are usually lowercase, while user-defined identifiers are capitalized throughout or in mixed-case. This difference in conventions can be addressed automatically when converting a Modula-2 system to Ada.

The type structure is simpler in Modula-2 than in Ada. Usually translating a Modula-2 system to Ada will result in simple data types that do not fully use Ada capabilities. Two aspects of Modula-2, however, will result in an Ada system with unnecessary complications. First, Modula-2 has no construct corresponding to the Ada generic. Thus, a Modula-2 system may have modules implementing closely related objects, such as a queue of integers and a queue of characters, with repeated code and no method imposed by the language showing the relation within these families of modules. A conversion effort should, at least, identify such families of modules and ensure each module is converted in a parallel manner. Consideration should be given to using an Ada generic to replace the entire family with one Ada generic package.

Second, Modula-2 "opaque types" may introduce unneeded complexity into the converted Ada system. The Modula-2 opaque types are made available in a definition module, but implemented in an implementation module. They correspond to Ada private types, and modules that import the type may only reference it by means of the operations exported from the module. Hence, Modula-2 can be used to enforce information hiding and data abstraction techniques (Vienneau 91). In many implementations, opaque types suffer from a limitation. The compiler must allocate space for a variable of an opaque type based solely on information in the module definition. No size information is given there since the full definition of the opaque type only appears in the module implementation. Consequently, many

implementations assume an opaque type is at most one word long. Modula-2 programmers overcome this restriction for objects, such as records, that require many words of storage by implementing such objects as pointers. Opaque types implemented as pointers can be mapped directly to Ada private and access types. Often, however, the complexity of dynamically allocated memory is not truly needed for the Ada implementation. Since the implementation of a private type in Ada is given in the private part of an Ada package specification, the specification provides the compiler with enough information to correctly allocate sufficient space to any variable of a private type. Consequently, those converting Modula-2 to Ada may want to include a step to simplify the resulting Ada code by removing unnecessary access types. This step probably cannot be performed automatically since it depends on semantic information about designers' intentions.

Probably the biggest problem with Modula-2 to Ada conversions involves system modules. Wirth provides a few "standard" modules for input-output, low-level system functions, and multitasking, but most implementations will provide many more. Any use of these system modules must be converted to corresponding Ada code. Most of these uses are not built into Ada. Input and output will be a problem, since the two languages differ in their implementation here. Ada input/output features are provided by generic instantiations of standard packages, while Modula-2 provides a separate module for input/output of each simple data type.

Not all Modula-2 programs use the tasking features of Modula-2, but those that do will be difficult to convert. The multiprocessing models of the two languages are very different. Modula-2 provides a separate module for concurrent processes; Ada has tasking embedded in the language. Modula-2 uses a more traditional use of signals for inter-process communication and synchronization, while Ada uses the task rendezvous features designed with the language. Perhaps the best method of converting a multitasking Modula-2 system to Ada is to first design an Ada package corresponding to the Modula-2 *Processes* module. This approach builds the Modula-2 tasking model on top of the Ada tasking model. This would be difficult, but the solution is perfectly general and would be capable of being reused in other Modula-2 to Ada conversions.

3.1.6 Pascal

Pascal is a third generation language originally designed by Niklaus Wirth for student use. It provides structured programming language constructs (sequence, selection, iteration), as well as powerful data structuring facilities. Because Pascal's initial intended use was in education, certain facilities that are convenient for industrial use were not provided. In particular, standard Pascal is particularly weak in Input/Output facilities, including file handling, and capabilities for creating and reusing libraries.

Stock (Stock) points out that due to the long delay in standardization, coupled with the tendency of compiler vendors to ignore the standards when they did appear, allowed for the creation of many variants of the language. As a result, virtually every Pascal compiler had its own extensions, limitations and variations.

The syntax of Ada is derived from Pascal, but Ada has many additional capabilities not provided in the Pascal standard. In particular, standard Pascal does not include any modularization capability above the subprogram level. A Pascal program is a single file containing a hierarchical structure of procedures and functions. No equivalents to Ada packages are provided. Consequently, a straight translation from Pascal to Ada is fairly simple, but the more modern features of Ada will not be used. The resulting Ada program will not be modularized as it would have been if it were originally developed in Ada. The information-hiding and object-oriented features of Ada will also not be used by a straight translation. Even in a line-by-line translation of Pascal, difficulties may arise in handling initialization, default values and discriminants when converting Pascal record constructs to Ada record constructs (Wallis 85).

Restructuring may well be needed for those few Pascal programs which make extensive use of the GOTO statement. Indeed for such a system any approach other than a complete redesign and rewrite may not be the best choice unless a tool which provides automatic restructuring, such as that discussed in Section 4.3, is used. A redesign and rewrite may also be desirable in the more general case in order to take full advantage of Ada's advanced modularization capabilities.

3.2 Previous Experiences and Results

Many advances have been made in the past 10 or so years in the field of conversion. This section attempts to summarize the main problems, and techniques for solving those problems, encountered by certain conversion efforts involving non-Ada source and Ada target systems since 1985. The following efforts are discussed:

- o WWMCCS ADP Component
- o WWMCCS UNITREP Subsystem of ADP Component
- o An Internal Kaman Sciences C-to-Ada Conversion Project

The third effort was performed in house by the author, and can therefore be discussed more fully. Appendix A and B contain the C-Interface data declarations and utility modules, respectively, which were developed during the effort.

3.2.1 WWMCCS ADP Component

Parsian (88) discusses the modernization of the Automated Data Processing (ADP) component of the Worldwide Military Command and Control System (WWMCCS) to the WWMCCS Information System (WIS) during the mid-1980s. The approach taken was to create a fully automated set of translation tools for performing the translation from Fortran 66 to Ada, resulting in a set of tools consisting of three components:

- o A Fortran Source Analyzer, which creates a tree of syntax for the Fortran code
- o A Fortran Tree Transformer, which transforms the Fortran tree into a similar tree for Ada source code
- o An Ada converter, which creates Ada source code from the second tree.

This is an example of the fifth approach described in Section 2.2, "Automatic Translation."

Parsian states the following strategy was used to map a Fortran program to Ada:

- o A procedure as a driver of the generated Ada code;

- o The main program and each Fortran subprogram translates into its own package with local variables declared in the package body (preserving the static nature of local variables in Fortran).

- o Functions and subroutines become procedures (expressed inside of packages) with "in out" arguments; the use of subprograms as parameters can be handled by use of the Ada generic units which have subprograms as generic formal parameters.

- o Each program's local non-EQUIVALENCEd variables can be stored in its own character array; and EQUIVALENCEd variables must be stored in the same character array, slices of which are renamed to relevant EQUIVALENCEd names.

- o COMMON blocks are translated as separate packages and imported by **with** and **use** clauses to the subprograms that contain this COMMON block structure (Parsian 88).

Parsian discusses and shows examples for converting most Fortran structures to Ada, including DO, FOR, IF, EQUIVALENCE, and COMMON. A very interesting aspect of this conversion effort was the treatment of Fortran data structures in COMMON and EQUIVALENCE structures. The approach to data type/structure conversion used on this project was to maintain all data in Ada string variables, and to implement a set of conversion modules which converted and manipulated the variables in Fortran format as they were used. The fact that data is maintained in Fortran format throughout the converted system, exemplifies the fact that Automatic Translation results in a converted system of Fortran coded in Ada and does not utilize the software engineering capabilities of Ada.

3.2.1.1 Fortran Equivalence Statements

The Fortran EQUIVALENCE capability was implemented by renaming strings over each other using the Ada slice mechanism. The following example is extracted from Parsian (88) to illustrate how Fortran EQUIVALENCE structures were emulated. Consider the following Fortran declarations:

```
REAL A( 5) , B( 6)
EQUIVALENCE ( A( 4) , B( 2) , N)
```

These statements imply that the fourth position of A, the second position of B and the integer N all occupy the same physical memory location. These Fortran statements would result in the following storage assignments:

```
memory: A(1) A(2) A(3) A(4) A(5)
                  B(1) B(2) B(3) B(4) B(5) B(6)
                       N
```

This would result in an Ada string variable containing 8 words. The WORD_SIZE was chosen to be 6 characters. The following Ada code was then generated to convert the above Fortran code::

```
EQUIVALENCE_STR:  STRING( 1 .. 8*WORD_SIZE) ;
A:                STRING
          renames EQUIVALENCE_STR( 1. . 5*WORD_SIZE) ;
B:                STRING
          renames EQUIVALENCE_STR( 2*WORD_SIZE+1..8*WORD_SIZE) ;
N:                FORT_INTEGER
          renames EQUIVALENCE_STR( 3*WORD_SIZE+1..4*WORD_SIZE) ;
```

FORT_INTEGER is a user defined string sub-type used by the conversion routines which manipulate Fortran variables and represents a Fortran integer data type. To facilitate conversion routines, an attribute record is created for each variable which contains the type of array, bounds of the array, element type for the array, and a pointer to the memory location for the first character of the variable.

Since all Fortran data types (real, integer, logical, complex, double precision, and character) are converted to Ada string variables, the equivalence mechanism can overload any combination of data types. Fortran equivalences differs from Fortran common blocks in that *EQUIVALENCE* assigns variables within the same program unit to the same physical memory location. *COMMON*, however, assigns variables in different program units to the same physical memory location.

3.2.1.2 Fortran Common Statements

Parsian's method of treating Fortran common blocks is most usefully explained by another example. This example is also extracted from Parsian (88). Suppose a Fortran main program and subroutine make use of the following common blocks:

Main program	**Subroutine**
COMMON /X/ B, A(5)	*COMPLEX P(2)*
COMMON /Y/ C(6, 7)	*COMMON /X/ P, I*
	COMMON /Y/ N(12)

The Fortran common capability is implemented by creating Ada package specifications for common blocks:

```
package COMMON_X is          -- COMMON /X/
    BLK:  STRING( 1 .. 6*WORD_SIZE );
    -- 6 = maximum{ (5 + 1), (2 * 2 + 1) }, where
    -- size_of(A)=5, size_of(B)=1, size_of(P)=2*2, size_of(I)=1.
end COMMON_X;

package COMMON_Y is          -- COMMON /Y/
    BLK:  STRING( 1 .. 42*WORD_SIZE );
    -- 42 = maximum{ (6 * 7), 12 }
end COMMON_Y;
```

The mapping for the main program will be as follows:

with COMMON_X; with COMMON_Y;
*B: STRING renames COMMON_X.BLK(1 .. 1*WORD_SIZE);*
*A: STRING renames COMMON_X.BLK(1*WORDSIZE+1 .. (5+1)*WORD_SIZE);*
*C: STRING renames COMMON_Y.BLK(1 .. 42*WORD_SIZE);*

The mapping for the subroutine will be as follows:

with COMMON_X; with COMMON_Y;
*P: STRING renames COMMON_X.BLK(1 .. 2*2*WORD_SIZE);*
I: FORT_INTEGER renames
*COMMON_X.BLK(2*WORD_SIZE+1 .. (1+2*2)*WORD_SIZE);*
*N: STRING renames COMMON_Y.BLK(1 .. 12*WORD_SIZE);*

Here again, to facilitate conversion routines, an attribute record is created for each variable, which contains the type of array, bounds of the array, element type for the array, and a pointer to the memory location for the first character of the variable.

3.2.2 WWMCCS UNITREP Subsystem of ADP Component

Dr. John Schill and co-authors (Shill 85) describe an Ada model for the conversion of the Military Command and Control System (WWMCCS) Unit Status and Resource Reporting Subsystem (UNITREP) within the WWMCCS Automated Data Processing (ADP) component. Prior to conversion, UNITREP was written almost entirely in Cobol with a few assembly language routines, which interface to a conventional database system for message storage and manipulation.

The Cobol system was estimated at approximately 300,000 lines of code, with the conversion resulting in an estimated 30,000 lines of Ada code (or 5000 Ada statements) (Shill 85). The first of our conversion approaches was used, "Complete Redesign and Rewrite," because the Cobol system worked with messages received as pseudo card images which were processed and maintained in a conventional sequential database. The history of the system with its many enhancements and modifications left much duplication and redundancy in both code and data. Therefore the authors do not claim

a 10:1 reduction in code is typical of Cobol-to-Ada conversions. Indeed, even a redesign in Cobol would have saved many lines of code (Shill 85).

The conversion approach taken made full use of Ada's software engineering capabilities including extensive use of data abstraction, generics, exceptions, and tasking for parallel processing of prioritized message queues. A relational database machine replaced the conventional sequential database system, allowing heretofore unavailable timely ad-hoc queries. The development staff found Ada to be reasonably easy to learn and use.

Speaking of the Ada model, the authors sum up with: "The advantages of this approach, in broad terms, are increased programmer productivity and a greatly improved product--more understandable, more maintainable programs, better able to grow and change with the new technologies of the 1990s and beyond" (Shill 85).

3.2.3 Internal Kaman Sciences C-to-Ada Conversion Project

An on-going internal technology transfer Research and Development project at Kaman Sciences Corporation involves converting two C program libraries to Ada. The system containing these libraries is an electronic forms and routing office automation system in which automated forms are used to store and retrieve data from a relational database system. As shown in Figure 3.2-1, the system being converted contains five main components:

- The Off-The-Shelf (OTS) Menu Package (MP),
- The User Interface (UI) written in 4GL,
- The Generic Library (LIBGEN),
- The Forms Processor (LIBFP) library,
- The Relational Database Server (RDS).

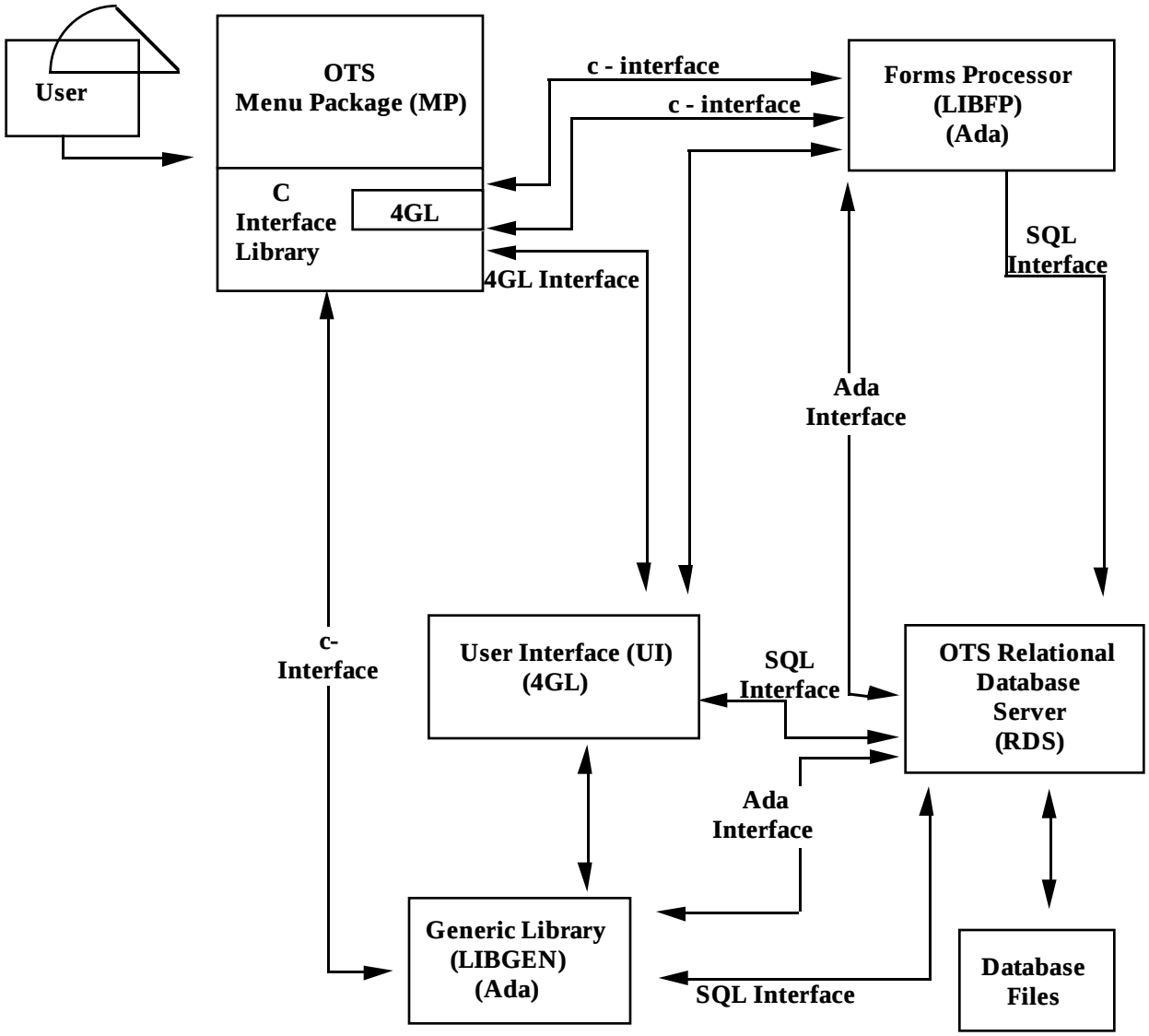

Figure 3.2 - 1 System Components

The system was developed by taking advantage of commercially available tools as much as possible. The OTS MP provides a nonprocedural fourth generation menu-oriented language (4GL), a screen manager, and a memory resident data dictionary which allow storage and manipulation of the data as it passes between the menu fields and the database files. The UI is a set of user developed menus and application oriented data flow modules written in 4GL. These modules have access to the user via the MP and to the database via Structured Query Language (SQL), LIBFP, and LIBGEN. The lines between the UI and the LIBGEN and between the UI and the LIBFP indicate that control and data pass through the 4GL interface. The 4GL instruction set supplies a "call" capability to both C and UNIX C-shell modules. The UI manipulates user screen-oriented forms via LIBFP. The forms are stored in the relational database as tables. The LIBFP uses the RDS system tables to build and manipulate screen versions of the forms. Forms processed by this system include multi-page forms used within an activity, with lines and boxes for dates, data, signatures, etc. The UI interfaces with the MP via the 4GL interface, and the LIBFP and LIBGEN interface with the MP via a C-library interface.

Originally, the LIBFP and LIBGEN were developed in C. The RDS was also accessed via a C-library interface. When it was decided to package this as a product for government agencies, however, it became apparent that Ada must be the development language. At that time the OTS RDS offered an Ada-interface, but the OTS MP did not. To change the MP to another vendor who offered an Ada-interface would require a complete rewrite of the UI 4GL code, which is the bulk of the product code. A concession was made, therefore, to use the MP with the C- interface, an Incremental Multi-lingual Rewrite approach for the LIBFP, an Incremental Functionally Equivalent Replacement with a Multilingual flavor for the LIBGEN, and the Ada interface for the RDS. The LIBGEN and LIBFP conversion approaches are discussed in the following paragraphs.

3.2.3.1 Generic Library (LIBGEN)

The Generic Library (LIBGEN) is a set of relatively independent modules with well defined interfaces. This library is an application-specific filter to the relational database system. The application itself was designed in such a way that the system is modified merely by altering the content of tables in the database. The "queuing" of

forms and packages of forms for routing though an office or organization is maintained as table data. In this way, senders and receivers of forms can be added to the queues by modifying database tables. Since the forms themselves are database tables, new forms can be added and current forms modified by the users, requiring no modification to the system. The LIBGEN library provides security protection over the tables. The security itself is also contained within tables, with designated users having controlled access to the security tables.

The LIBGEN was well designed, structured, and modularized. The implementers were also trained in Ada, so an object oriented flavor, with some data hiding, was used in the design of the C modules. This all led to the decision to use an Incremental Functionally Equivalent Replacement approach for the conversion for LIBGEN. Since LIBGEN had to continue to use the C-interface library to interface with the MP, a multilingual flavor was added to the conversion. This meant that the converted Ada modules could interface to the RDS via the new Ada-interface, but must use "parallel data types" when called from the 4GL UI. Ada modules which call screen management routines must also use the MP C-interface and "parallel data types." It was for this reason that the *C_Interface* package, listed in Appendices A and B, was created.

The Incremental Functionally Equivalent Replacement approach allowed the LIBGEN modules to be replaced inline with the production system. All testing was accomplished using the system itself, with the aid of the source-code debugger. The system executable was linked first with the newly built Ada Library, then with the RDS Ada-interface, then with the MP C-interface, and finally with the original C-version of LIBGEN.

In order to use the Ada software engineering capabilities such as exceptions, elaboration and generics, the "main" routine had to be written in Ada and the executable had to be linked by the Ada loader to the Ada Run-time system. A problem was encountered at this point because the MP supplies a "main" routine which maintains information and overall control over the resultant executable. The application developer is free to develop a new "main" routine. This approach was decided against when the system was originally designed. Rather than step back and develop a new Ada "main," a dummy Ada "main" was built which accepts the command line arguments and passes them to the original C "main" routine as a subroutine.

Once all of this was decided, the task of creating C-to-Ada parallel data types was begun, the results of which are listed in Appendices A and B. Appendix A contains the C interface package specification, which contains five basic parts:

- o A set of parallel data types
- o A set of *UNCHECKED_CONVERSION* functions to help in parallel data type conversions from C-to-Ada and Ada-to-C
- o A set of interface declarations for the MP screen manager functions / procedures
- o A set of declarations for conversion support functions / procedures which are in the package body
- o A set of interface declarations for functions / procedures in the UNIX system library

The SUN Ada 1.0 compiler and toolset, based on the Verdix Corporation Verdix Development System, Version 6.0, was used for the conversion. The parallel data types in the C-interface Package were developed as suggested in "Interface Programming," Chapter 4 of the *Sun Ada Programmer's Guide*. It is quite possible, therefore, that modifications to these parallel data types might be required if another vendor's Ada development system is used.

The *UNCHECKED_CONVERSION* functions allow direct conversion between data types which can be overloaded due to the fact that we are using the same hardware for the C and Ada modules. C strings are passed to and received from the Ada modules as a *SYSTEM.ADDRESS* data type, as defined in the *SYSTEM* Ada package.

An example of this is in the declaration of the interface to the *sm_emsg* screen manager procedure. Over 50 screen manager functions were actually in the C interface package, but were removed for brevity. Note that in the procedure *get_next_arg* in Appendix B, there is a debug statement, which has been commented out, that uses the

sm_emsg procedure to display a message on the screen. *my_mesg* is a standard Ada string, and is used only for this purpose. The *sm_emsg(my_mesg(1)'address)* statement is an example of passing a C string data type to the MP C interface. Note that the message string is terminated with an *ASCII.NUL* to meet the C string requirements, and also that the *arg_str* received from the 4GL call is a standard C string terminated with a null. Other examples of Ada-to-C calls with parallel data types are the declarations of *crypt*, *getlogin* and *getpid*.

The LIBGEN C library contained 79 procedures comprising 6230 lines of C code. The converted LIBGEN Ada library was collected into 8 packages containing 78 Ada subprograms comprising 6702 lines of Ada code. This effort was conducted by one person over six months and consumed 668 hours, or one third of a man year.

3.2.3.2 Forms Processor (LIBFP)

The Forms Processor (LIBFP) is a set of closely coupled modules which perform a set of services for the 4GL code. While the LIBGEN modules are individually called from the 4GL code, LIBFP is called at a common entry. This means that there is only one point where "parallel data types" are required as input parameters. The LIBFP does make use of the screen manager, however, and must use "parallel data types" to interface with the MP.

Forms, as described earlier, are stored in the database as tables. When the user wishes to interact with a form, LIBFP interfaces to the RDS at the system table level to extract the row/column type and size information for the table(s) in which the form is stored. This information is used to create a softcopy of the form on the user's screen. The form is stored in dynamic memory as a set of linked lists, requiring extensive dynamic memory allocation and deallocation. The memory management for LIBFP was originally performed using the C *malloc*, *calloc*, *free,* and *cfree* system interfaces.

The first task, therefore, was to design a memory management mechanism under Ada. Most Ada memory management schemes involve some sort of heap mechanism which is usually platform dependent. We decided that a more portable scheme would suit our marketing plans. Since the forms are mostly text, a dynamic string mechanism would also be needed. In the course of one session a user might require 20 or more forms to be dynamically built and displayed. Using Ada's string data type would not

support the hundreds of dynamic strings of lengths varying from 10 characters to 6 or 7 thousand characters for multiple pages of text. The dynamic strings are linked together by 5 types of dynamically created headers, each header containing pointers to from one to several strings.

A generic memory manager was created along the lines of one made popular by Grady Booch in the early to mid 1980s. The memory manager maintains a list of free space. When processing a request for additional space, the manager will either return the top item from the free list, or if the free list is empty will create and return a new item via the Ada "new" mechanism. When an item is returned to the memory manager, it is added to the top of the free list for later allocation.

By "instantiating" a memory manager for each of the five header types, each of the five free lists contain like items. As the first form is being built, all headers will be '"new" and will be allocated from the system heap by Ada. When the form is deleted, all headers will be returned to the appropriate free lists to be used by the next form. Memory allocation will be high at the start of a session, but will remain fairly static for the bulk of the session because all forms have basically the same header requirements.

The problem of dynamic strings was another matter, yet was solved in basically the same way as were the headers. A dynamic string package was chosen from a previous government effort at Rome Laboratory (RL) in Rome, New York. RL has given permission to make the technology transfer to our effort. The dynamic string package was also created along the lines of one made popular by Grady Booch. The package makes use of the memory manager to create the dynamic strings. The Booch version, however, instantiated a new memory manager for each new dynamic string length, maintaining a free list for each length. New strings were then requested from a memory manager whose string lengths were equal to or greater than the length of the required string.

A RL Software Engineer decided to improve the package by changing the dynamic strings to a header with a linked list of "nodes" of characters. The package became a generic package for which the caller supplied the node size as a generic parameter, and which returned an empty header. Part of the instantiated package is an instantiation of the memory manager for items of node size length. The package also contains a set of string operations which can be applied to a dynamic string. The first

operation usually performed is to allocate a length to the string. This string length is divided by the node size, and one more than that number of nodes are requested from the memory manager, are link listed, and are hung from the header. The string header contains pertinent information about the string, such as a pointer to the first node, the allocated length of the string, and the current actual length of the string. When the string is deleted, using one of the string operations, the nodes are returned to the free list via the memory manager and the header is modified to reflect a zero allocated string length. The caller is then free to allocate a new length to the string (header) for a different text string. Headers are created with the Ada "new" mechanism and never disappear.

Since 90% of the dynamic strings used within the LIBFP are hung from one of the five headers previously described, pointers to dynamic strings were more useful than the headers themselves. A further modification was therefore made to the dynamic string package to return a pointer to a dynamic string header rather than the header itself. This involved adding a second instantiation of the memory manager to the dynamic string package. The first memory manager is dubbed the "node manager," and the second memory manager is dubbed the "string manager." When a new string is now requested, a string header is requested from the string manager and its pointer is returned to the caller. The string operations were then modified to accept either a string header or a pointer to a string header. This added the benefit that dynamic string headers were also reused during a session.

The RL development Software Engineer determined that a node of 10 characters worked well for their effort. Figure 3.2-2 depicts the free lists, an allocated empty header, and an allocated dynamic string with length of 12 characters using a node size of 10. Analysis must yet be performed to determine the optimum "node size" for the LIBFP.

Figure 3.2-3 depicts an abbreviated memory layout for a form generated and manipulated by the LIBFP. The Form, Row and Column Headers are 3 of the 5 headers described earlier. The Form Header contains a pointer to a dynamic string containing the name of the form, a pointer to the first Row Header, and a pointer to the next Form Header if more than one form is being worked on at the time. Each Row Header contains a pointer to the first Column Header and a pointer to the next Row Header. Each Column Header contains a pointer to a dynamic string containing the name of the

column, a pointer to a dynamic string containing the old column value if it has been modified since display, a pointer to a dynamic string containing the current column value, and a pointer to the next column in the row. The headers actually contain more data such as the database type of the column, etc., and other headers allow repeating columns and scrolling arrays of columns.

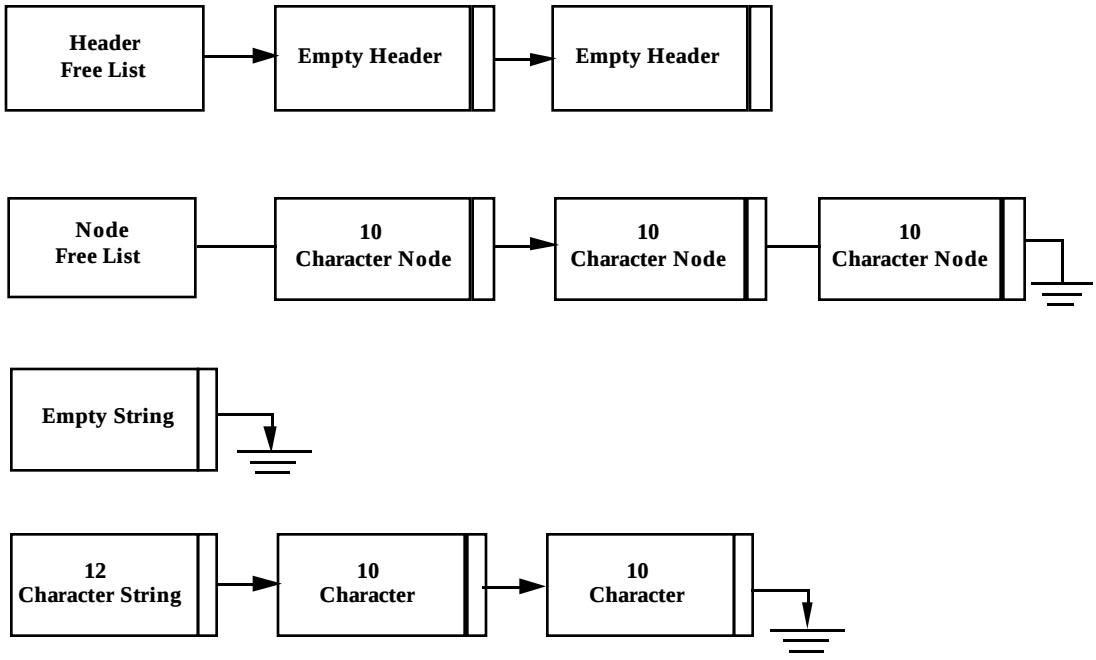

Figure 3.2 - 2 Dynamic Strings with Node = 10

Since the memory management and dynamic strings were completely redone, an Incremental Multilingual Rewrite approach was adopted for the LIBFP. The C modules will be functionally replaced; there will probably be one new Ada module for each old C module, but each module will be redesigned to make good use of the memory management and dynamic strings and may not closely resemble the original module. It is not clear at this time if there will be the same number of modules in the converted system. The redesigned modules, however, will still have to interface with the screen manager, accounting for the multilingual flavor.

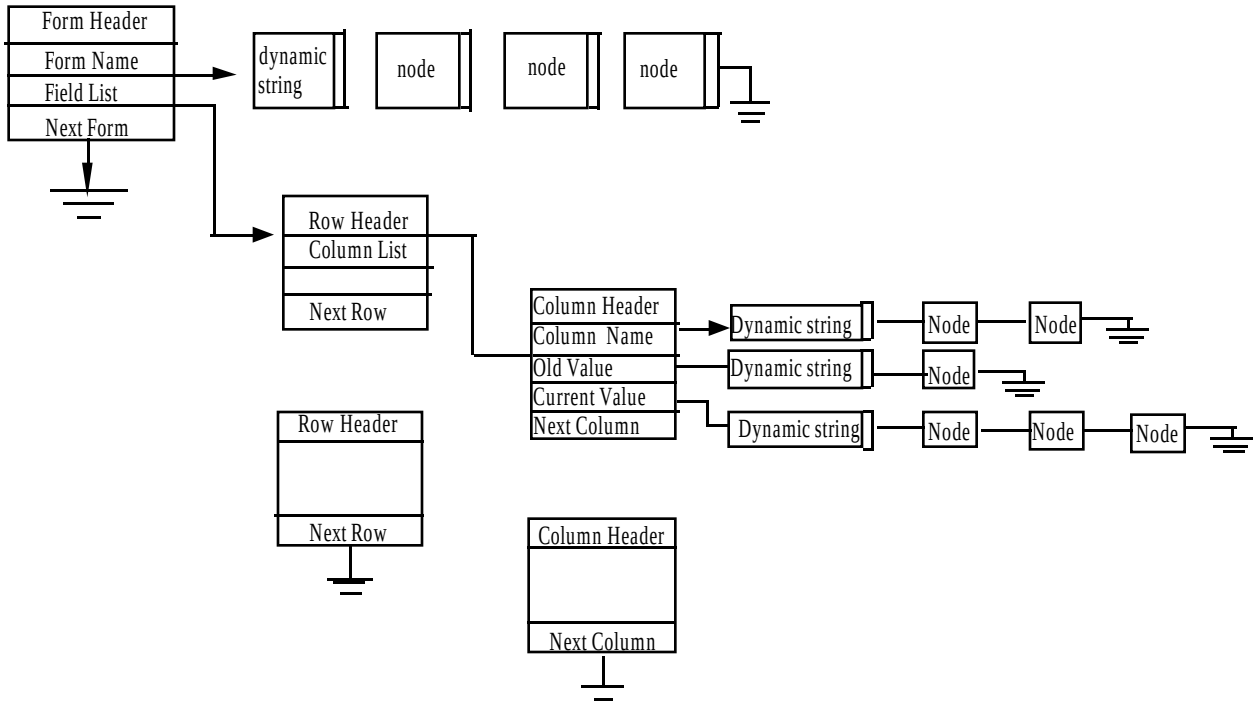

Figure 3.2-3 Abbreviated Form Layout

The LIBFP C library contained 102 modules comprising 5281 lines of C code. The LIBFP Ada library is still under conversion. The memory management subprograms, which have been completed, are collected into three packages containing five subprograms comprising 1013 lines of Ada code. The bases of these packages were reused from the RL effort with perhaps 30% modification to tailor them to this effort.

4. CONVERSION AIDS AND TOOLS

The vendors discussed in this section were asked for information sufficient to include their products in this paper as a reference for those managers who would like to investigate currently available automatic translation tools. Only those vendors who responded are included here. Not all who responded supplied the same amount or type of information, making it hard to do any comparisons.

No conclusions or recommendations are provided as to which vendor or vendor product is better than another. Automatic conversions tools available from three vendors are described here. The three vendors are:

- Applied Conversion Technologies, Inc.
  415 U.S. Highway One
  Suite C
  Lake Park, FL 33403
  (407)844-0000

- R.R. Software, Inc.
  PO Box 1512
  Madison, Wisconsin 53701
  (608)251-3133

- Scandura Intelligent Systems
  1249 Greentree
  Narberth, PA 19072
  (215)664-1207

4.1 Applied Conversion Technology

Applied Conversion Technology has expert system-based tools which directly support conversion from Fortran, assembly and C to Ada, with Cobol-to-Ada under development. Their technology, called "Transformer" or "X4MR" is based on the domain-dependent X4MR Expert System Shell, and offers support for other language conversions as well as for system migration, development, and maintenance. Conversion and support products are X4MR RuleBase driven, and operate in a

Workstation (386/486) and MS-DOS environment. X4MR was written in C for platform flexibility.

No information was supplied describing the RuleBase and there were no available samples of source and target programs. Resultant target programs are described to be functionally equivalent to the source programs. The X4MR RuleBase implies, however, that the RuleBase developer could enhance the conversion using the expert-system-based tools. The vendor licenses the technology, tools, and consulting services to organizations which are in the business of doing conversions for their clients (Seifert).

4.2 R. R. Software

R. R. Software has a tool, PasTran 2.0, which converts Pascal source code to Ada source code and is a compromise between simple syntactic translation and 100 percent automatic translation. PasTran itself is written in Ada, the vendor being the developer of the JANUS/Ada compiler. Source programs are required to be in ANSI Standard Pascal. The approach taken was that the code is incompletely translated, leaving some hand coding for the person desiring the translation. The untranslated portion is concerned with the various dialects of the several Pascal compilers. Since Pascal was the forerunner of Ada, the general flavor of the languages is similar thus avoiding many of the problems encountered with other source languages such as Fortran or Cobol (Stock).

4.3 Scandura Intelligent Systems

Scandura Intelligent Systems has three tools which directly support conversion from non-Ada source code to Ada source code. Several other tools support design and implementation using the intermediate source format, FLOWforms. Languages supported are C, Cobol, and Fortran. Each language is supported by a re/Nu Conversion Workbench which uses a standard set of PRODOC components.

The conversion process consists first of restructuring the source language into a set of call hierarchy and module FLOWforms. Tools support manipulation of FLOWforms from simple editing to actual enhancement of the original design. Once the source language code has been restructured into FLOWforms, and

modified/enhanced as desired, these FLOWforms are then converted to Ada FLOWforms. This process moves the essence of the Source language code to a functionally equivalent Ada essence. An Ada program is then generated from the Ada FLOWforms.

Scandura explains the theory of FLOW form creation: "This cognitive approach involves modeling and testing the structural and functional essence of a system at a high level of abstraction, with increasing specificity until contact is made with available data and computational resources" (Scandura 93). Vendor documentation shows examples of how data and modules are restructured into FLOWforms. Language specific problems, such as converting Fortran data formats to Ada data formats, Fortran and Cobol GOTOs, etc., are handled. Globals are handled by separate Ada packages. The vendor documentation, however, does not mention if or how Fortran EQUIVALENCE statements are supported. This process offers a much needed capability, the removal of GOTOs via restructuring. The extent to which this can be accomplished on actual old Fortran and Cobol code will be the test of its usefulness in non-Ada to Ada conversion.

The vendor claims that well structured non-Ada code can be translated to Ada in a matter of hours where the process would take many man-years of effort by hand. Although the resultant Ada code is actually a rewrite of the original source code, i.e., a Fortran program written in Ada, the tools available to manipulate the FLOWforms offer an opportunity to slip in good Ada capabilities where they would be most needed after the Ada FLOWforms are created and before the Ada code is generated.

The re/Nu Conversion Workbench family operates in an MS-DOS PC-AT (640K RAM) environment, providing a comprehensive but simple and uniform means of graphically representing data and processes during all phases of the conversion process. Conversion of source code from one language to another is not the prime purpose for the tool set offered by this vendor. The tools are designed to be used throughout the life cycle of a system, from new systems to partially re-engineered systems to totally re-designed systems. The tools offer a full spectrum of life cycle documentation. The vendor offers an engineering service/trial program to reverse engineer a sample of the customers source code and return it with a functional demo program for the customer to see the results (Scandura 87).

5. SUMMARY

In the past five to seven years Software Engineering advances have made thousands of production systems, in both government and industry, outdated and obsolete. Decisions must be made to upgrade these systems by redesign and development or conversion. A DoD mandate (DoD Directive 3402.1) has made Ada the target in all such decisions concerning the Government. This also filters down to Software Contractors and Software Product Developers who wish to get or keep the Government as a customer in the future.

There are various approaches to the conversion process, including:

- o Complete Redesign and Rewrite
- o Incremental Functionally Equivalent Replacement
- o Incremental Redesign and Rewrite
- o Incremental Multilingual Rewrite
- o Automatic Translation

Each approach has its advantages and disadvantages, and the approach taken must be chosen based upon such criteria as ease of conversion from the source HOL to Ada, the available resources, and the currency and validity of the source system design and specifications.

When approaching a conversion effort, changes must be expected and made in both the management and technical expertise of the enterprise, including upper level management.

Sufficient homework must be done up front, or the conversion effort will fail. This includes a set of steps which will prepare all of the data and conversion aid tools that will be needed by the conversion team. Each source HOL has its own problems when being converted to Ada. These problems must be recognized early in the conversion effort, and concrete plans must be made for their resolution.

There are others out there with similar conversion problems, and an effort should be made to locate them. Their conversion team is sure to have documented their effort, and this documentation could save your effort a lot of false starts and problems.

New OTS tools to help with the conversion process are coming onto the market. They should be looked at closely at the start of your conversion effort. They may not be a complete answer, but their cost could be less than the amount of resources they could potentially save.

6. REFERENCES

(Ehrenfried 82) P. J. L. Wallis, "Feasibility Assessment of JOVIAL to Ada Translation", *Proceedings of 2nd AFSC Standardization Conference*, Vol. 1, 30 Nov - 2 Dec 1982.

(Horton 85) Michael J. Horton and Teri F. Payton, "Integrating Ada into Multi-lingual Systems: Issues and Approaches", *Proceedings of the 3rd Annual National Conference on Ada Technology*, 1985.

(Kernighan 88) Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second Edition, Prentice Hall, 1988.

(Martin 85) Donald G Martin, "Non-Ada to Ada Conversion", *Journal Of Pascal, Ada, & Modula-2*, Vol. 4, No. 6, pp. 36-40, Nov / Dec 1985.

(Parsian 88) Mahmoud Parsian, "Ada Translation Tools Development: Mappings From Fortran To Ada", *The 3rd International IEEE Conference on Ada Applications and Environments*, pp. 117-135, May 1988.

(RM 83) *Reference Manual for the Ada Programming Language*, ANSI / MIL-STD-1815A, January 1983.

(Scandura 87) Joseph M. Scandura, "A Cognitive Approach to Software Development: The PRODOC Environment and Associated Methodology", *Journal of Pascal, Ada & Modula-2*, Vol. 6 No. 5, 1987.

(Scandura 93) Joseph M. Scandura, “Automating Renewal and Conversion of Legacy Code into Ada: A Cognitive Approach", Scandura Intelligent Systems, 1993.

(Seifert) Ralph W. Seifert, "X4MR-Based Automated Code Transformation", Vendor Marketing Literature, no date.

(Shumate 84) Kenneth C. Shumate, *Understanding Ada*, Harper & Row, 1984.

(Shill 85) John Shill, Roger Smeaton, Richard Jackman, "The Conversion of Command & Control Software to Ada: Experiences and Lessons learned", *ACM Ada Letters*, Vol. IV, Issue 4, Jan / Feb 1985.

(Stock) Daniel L. Stock and James A. Stewart, "From Pascal to Ada: A Practical Perspective on Translation", Abstracted From *IEEE Computer*, supplied by R.R. Software, Inc., no date.

(Vienneau 91) Robert Vienneau, *An Overview of Object Oriented Design,*, Data & Analysis Center for Software, April 30, 1991.

(Wallis) P. J. L. Wallis, "Automatic Language Conversion and its Place in the Transition to Ada", *Special Edition of Ada Letters*, Vol. V, Issue 2, Sep / Oct 1985.

(Wirth 83) Niklaus Wirth, *Programming in Modula-2*, Second Edition, Springer-Verlag, 1983.

Appendix A. C-Interface Parallel Data Types

```
------------------------------------------------
--                    PACKAGE SPECIFICATION
--
--
-- PACKAGE : c_interface_.a
-- VERSION  : 1.0
-- WRITTEN  : 09/18/92
-- SCCS FILE :
--
-- DESCRIPTION:
--
--      This is an Ada package which contains the parallel data types
-- and functions required to interface between a main Ada program and
-- a C subroutine library.  The procedure and function bodies are in
-- c_interface.a.
--
-- PACKAGES WITH'D:
--
--      SYSTEM
--      UNCHECKED_CONVERSION
--      LANGUAGE
--      UNSIGNED_TYPES
--
-- PACKAGES USE'D:
--
-- WITH'D BY:
--
--      With'd by every LIBGEN routine.
--
-- ROUTINES IN PACKAGE:
--
--      address_to_c
--      c_to_address
--      integer_to_address
```

```
--      address_to_integer
--      integer_to_c
--      c_to_integer
--      integer_to_c_u
--      c_u_to_integer
--      c_string_to_integer
--      get_next_arg
--      c_length
--      string_to_c
--      allocate_c
--      deallocate_c
--
-- UNIX ROUTINES USED:
--
--      crypt
--      free
--      getenv
--      getlogin
--      getpid
--      longjmp
--      malloc
--      setjmp
--      signal
--      sleep
--      system
--
-- MODIFICATIONS:
--      Data            Modifier        OS Version                      Modification
--      -----           ------------ ------------------------------------
--
-------------------------------------------------------------------------
--

with SYSTEM;
with UNCHECKED_CONVERSION;
with LANGUAGE;
```

```
with UNSIGNED_TYPES;

package C_INTERFACE is

        pragma SUPPRESS(ALL_CHECKS);

  --
  -- The following type definitions supply the parallel data types
  -- needed to interface between C and Ada routines.
  --
  type c_string is access string (1..INTEGER'LAST);

  type c_int is range -( 2**31 )..( 2**31 )-1;
  for c_int'size use 32;

  type c_long is range -( 2**31 )..( 2**31 )-1;
  for c_long'size use 32;

  type c_short is range -( 2**15 )..( 2**15 )-1;
  for c_short'size use 16;

  type c_byte is range -( 2**7 )..( 2**7 )-1;
  for c_byte'size use 8;

  type c_char is range 0..( 2**8 )-1;
  for c_char'size use 8;

  type c_unsigned_int is new UNSIGNED_TYPES.UNSIGNED_INTEGER;

  type c_unsigned_short is range 0..( 2**16 )-1;
  for c_unsigned_short'size use 16;

  type c_bool is new boolean;
  for c_bool'size use 8;

--
```

```
-- The following procedures and functions supply various string
-- manipulation capabilities needed to work with both C and Ada
-- string types.  The procedure and function bodies are in
-- the c_interface.a package body.
--

function address_to_c is new UNCHECKED_CONVERSION(
    SOURCE => system.address,
    TARGET => c_string);

function c_to_address is new UNCHECKED_CONVERSION(
    SOURCE => c_string,
    TARGET => system.address);

function integer_to_address is new UNCHECKED_CONVERSION(
    SOURCE => integer,
    TARGET => system.address);

function address_to_integer is new UNCHECKED_CONVERSION(
    SOURCE => system.address,
    TARGET => integer);

function integer_to_c is new UNCHECKED_CONVERSION(
    SOURCE => integer,
    TARGET => c_int);

function c_to_integer is new UNCHECKED_CONVERSION(
    SOURCE => c_int,
    TARGET => integer);

function integer_to_c_u is new UNCHECKED_CONVERSION(
    SOURCE => integer,
    TARGET => c_unsigned_int);

function c_u_to_integer is new UNCHECKED_CONVERSION(
    SOURCE => c_unsigned_int,
```

```
    TARGET => integer);

function c_string_to_integer is new UNCHECKED_CONVERSION(
    SOURCE => c_string,
    TARGET => integer);

function integer_to_c_string is new UNCHECKED_CONVERSION(
    SOURCE => integer,
    TARGET => c_string);

  procedure get_next_arg (arg_str: in c_string;
                          arg:     in out c_string;
                          index:   in out natural);

function c_length(in_str: in c_string) return integer;

function c_to_string(in_str: in c_string) return string;

function string_to_c(in_str: in string) return c_string;

function allocate_c(len: c_unsigned_int) return c_string;

procedure deallocate_c(in_str: c_string);

--
-- The following are the interface declarations for the screen
-- manager library.
--

  procedure sm_emsg (error_msg : system.address);
  pragma INTERFACE (C, sm_emsg);
  pragma INTERFACE_NAME (sm_emsg,
                                    LANGUAGE.C_SUBP_PREFIX & "sm_emsg");

  ...
```

```
--
-- The following procedures and functions are from the UNIX system
-- library.  These routines were used since there was already a
-- mixture of C and Ada code being used.
--

function crypt (trigger, salt: system.address) return c_string;
   pragma INTERFACE (C, crypt);
   pragma INTERFACE_NAME (crypt, LANGUAGE.C_SUBP_PREFIX & "crypt");

function free (in_str: system.address) return integer;
   pragma INTERFACE (C, free);
   pragma INTERFACE_NAME (free, LANGUAGE.C_SUBP_PREFIX & "free");

function getenv (in_str: system.address) return c_string;
   pragma INTERFACE (C, getenv);
   pragma INTERFACE_NAME (getenv, LANGUAGE.C_SUBP_PREFIX & "getenv");

function getlogin return c_string;
   pragma INTERFACE (C, getlogin);
   pragma INTERFACE_NAME (getlogin, LANGUAGE.C_SUBP_PREFIX & "getlogin");

function getpid return c_int;
   pragma INTERFACE (C, getpid);
   pragma INTERFACE_NAME (getpid, LANGUAGE.C_SUBP_PREFIX & "getpid");

   procedure longjmp (env : system.address; val : c_int);
   pragma INTERFACE (C, longjmp);
   pragma INTERFACE_NAME (longjmp, LANGUAGE.C_SUBP_PREFIX & "longjmp");

function malloc (size: c_unsigned_int) return c_string;
   pragma INTERFACE (C, malloc);
   pragma INTERFACE_NAME (malloc, LANGUAGE.C_SUBP_PREFIX & "malloc");

   procedure signal (sig : c_int; func : system.address);
   pragma INTERFACE (C, signal);
```

```
   pragma INTERFACE_NAME (signal, LANGUAGE.C_SUBP_PREFIX & "signal");

function setjmp (env : system.address) return c_int;
   pragma INTERFACE (C, setjmp);
   pragma INTERFACE_NAME (setjmp, LANGUAGE.C_SUBP_PREFIX & "setjmp");

   procedure sleep (seconds : c_unsigned_int);
   pragma INTERFACE (C, sleep);
   pragma INTERFACE_NAME (sleep, LANGUAGE.C_SUBP_PREFIX & "sleep");

   procedure unix_system (in_string : system.address);
   pragma INTERFACE (C, unix_system);
   pragma INTERFACE_NAME (unix_system,
                                    LANGUAGE.C_SUBP_PREFIX & "system");

end C_INTERFACE;
```

Appendix B. C-Interface Utilities

```
------------------------------------------------------------------
--                                  PACKAGE BODY
-
-- PACKAGE : c_interface.a
-- VERSION          : 1.0
-- WRITTEN :        :09 / 21 / 92
-- SCCS FILE :
-
-- DESCRIPTION:
-
-       This is the body for the c_interface package.
-
--PACKAGES WITH'D:
-
-       SYSTEM
-
--PACKAGES USE'D:
-
-       C_INTERFACE
-
--WITH'D BY:
-
-- ROUTINES IN PACKAGE:
-
-       get_next_arg
-       c_length
-       string_to_c
-       allocate_c
-       deallocate_c
-
-- MODIFICATIONS:
-       Date            Modifier OS Version                 Modification
--      -------- -       ------- ----------- ------------------------------------------
-
-------------------------------------------------------------------------------------------
-
with SYSTEM;

package body c_interface is

        pragma SUPPRESS(ALL_CHECKS);

        my_mesg : string(1..256);                   -- debug message buffer
```

```
--------------------------------------------------------------------------------
--                                PROCEDURE
--
-- Copyright (c) 1992
--        Kaman Sciences Corporation
--        258 Genesee Street
--        Utica, New York  13502-4627
--
-- MODULE:  get_next_arg
-- PACKAGED IN:  c_interface
-- SEPARATE OF:
-- VERSION:  1.0
-- WRITTEN:  09/18/92
-- SCCS FILE:
--
-- DESCRIPTION:
--
--        This is a string manipulation routine designed to parse
-- 'space' separated 'words' from the parameter string passed to
-- C(Ada) functions by 4GL 'call' statements.
--
-- SEPARATES:
--
-- PACKAGES WITH'D:
--
--        None.
--
-- PACKAGES USE'D:
--
--        None.
--
-- ROUTINES CALLED:
--
-- CALLED BY:
--
-- CALLING SEQUENCE:
--
--   arg_str:  c_string;          -- from 4G
--   arg       :  c_string;       -- the name of this routine
--   index   :  natural := 1;    -- an index into the param string
--
--   get_next_arg(arg_str, arg, index);
--
--        The procedure will begin parsing at 'index', and will
--
```

```
--      1.  skip over spaces in 'arg_str' to the first non-space character,
--      2.  copy non-spaces characters into 'arg' until
--          a space is encountered,
--      3.  update 'index' to point to the newly found space.
--
--      The 'arg' c_string variable will be dynamically allocated,
--      and should be deallocated using the 'deallocate_c' function
--      as soon as it is no longer required.
--
- MODIFICATIONS:
--      Date            Modifier OS Version                 Modification
--      --------------- ---------- ---------------------------------------
--
-------------------------------------------------------------------------------
--

procedure get_next_arg (arg_str: in c_string;
                        arg      :   in out c_string;
                        index    :  in out natural) is

   local_index: natural;
   i            : natural;
   tmp_str      : string(1..255);

begin

--my_mesg := "GET_NEXT_ARG: index = " & natural'image(index) &
--          , arg_str = "" & arg_str(1..c_length(arg_str)) & """ & ASCII.NUL;
--sm_emsg(my_mesg(1)'address);
  local_index := index;
  i := 1;

  while arg_str(local_index) = ' '  and
              arg_str(local_index) /= ASCII.NUL loop      -- skip white space
    local_index := local_index + 1;
  end loop;

  while arg_str(local_index) /= ' ' and
              arg_str(local_index) /= ASCII.NUL loop      -- extract next arg
    tmp_str(i) := arg_str(local_index);
    local_index := local_index + 1;
    i := i + 1;
  end loop;
  tmp_str(i) := ASCII.NUL;              -- terminate with NULL
  index := local_index;                 -- update caller's index
```

```
   arg := allocate_c(c_unsigned_int(i));       -- allocate a c_string
   arg(1..i) := tmp_str(1..i);                 -- copy extracted arg
                                               -- to the c_string
end get_next_arg;

--
-------------------------------------------------------------------------------
--                                  FUNCTION
--
-- MODULE:  c_length
-- PACKAGED IN:  c_interface
-- SEPARATE OF:
-- VERSION:  1.0
-- WRITTEN:  09/21/92
-- SCCS FILE:
--
-- DESCRIPTION:
--
--        This is a string manipulation routine designed to compute
-- the length of a type 'c_string' variable.  Returned is the
-- number of characters which appear before the ASCII.NUL character.
--
-- SEPARATES:
--
-- PACKAGES WITH'D:
--
--        None.
--
-- PACKAGES USE'D:
--
--        None.
--
-- ROUTINES CALLED:
--
-- CALLED BY:
--
-- CALLING SEQUENCE:
--
--   in_str : c_string;
--   length : integer;
--
--   length := c_length(in_str);
```

```
--
-- MODIFICATIONS:
--       Date            Modifier OS Version                  Modification
--       --------  ------- ----------- ------------------------------------------
--
-------------------------------------------------------------------------------
--
--
function c_length(in_str: in c_string) return integer is
begin
  for i in in_str'range loop
    if in_str(i) = ASCII.NUL then
      return i-1;
    end if;
  end loop;
  return 0;
end c_length;

-------------------------------------------------------------------------------
--                                  FUNCTION
--
-- MODULE:  string_to_c
-- PACKAGED IN:  c_interface
-- SEPARATE OF:
-- VERSION:  1.0
-- WRITTEN:  09/22/92
-- SCCS FILE:
--
-- DESCRIPTION:
--
--       This is a string manipulation routine designed to convert
-- a standard 'string' variable to a type 'c_string' variable.
--
-- SEPARATES:
--
-- PACKAGES WITH'D:
--
--       None.
--
-- PACKAGES USE'D:
--
--       None.
--
-- ROUTINES CALLED:
--
-- CALLED BY:
```

```
--
-- CALLING SEQUENCE:
--
--   in_str  : string(1..n);
--   out_str : c_string;
--
--   out_str := string_to_c(in_str);

--
- MODIFICATIONS:
--        Date            Modifier OS Version                  Modification
--        -------- -       ------- ----------- ------------------------------------------
--
-------------------------------------------------------------------------------------------
--


--
--
function string_to_c(in_str: in string) return c_string is
  c_tmp : c_string;
  i     : natural;
begin
  i := in_str'last;                                  -- length of in_str.
  c_tmp:=allocate_c(c_unsigned_int(i + 1));          -- allocate c_tmp,
  c_tmp(1..i) := in_str(1..i);                       -- adding room for NULL.
  c_tmp(i + 1) := ASCII.NUL;                         -- copy in_str to c_tmp
  return c_tmp;                                      -- and add the NULL.
end string_to_c;                                     -- return c_string to
                                                     -- the caller.
--
-----------------------------------------------------------------------------
--                                  FUNCTION
--
-- MODULE  :  allocate_c
-- PACKAGED IN   :  c_interface
-- SEPARATE OF   :
-- VERSION       :  1.0
-- WRITTEN       :  09/22/92
-- SCCS FILE :
--
-- DESCRIPTION:
--
--        This is a string manipulation routine designed to allocate
-- room for a type 'c_string' variable.  This is done via the UNIX
-- system routine 'malloc'.
--
```

```
-- SEPARATES:
--
-- PACKAGES WITH'D:
--
--        None.
--
-- PACKAGES USE'D:
--
--        None.
--
-- ROUTINES CALLED:
--
--        malloc
--
-- CALLED BY:
--
-- CALLING SEQUENCE:
--
--   length  : c_unsigned_int;
--   out_str : c_string;
--
--   out_str := allocate_c(length);
--
--
- MODIFICATIONS:
--        Date             Modifier OS Version                  Modification
--        ---------------- ---------- --------------------------------------
--
-------------------------------------------------------------------------------------------
--
function allocate_c(len: c_unsigned_int) return c_string is
  local_len : c_unsigned_int;
  local_str : c_string;
begin
  if (len > 0) then
    local_len := ((((len - 1) / 4) + 1) * 4);   -- round to 4-bytes
  else
    local_len := 4;
  end if;
  local_str := malloc(local_len);
  local_str(1) := ASCII.NUL;
  return local_str;
exception
  when others =>
    my_mesg := "ALLOCATE_C: memory allocation error" & ASCII.NUL;
    sm_emsg(my_mesg(1)'address);
```

```
    return integer_to_c_string(0);
end allocate_c;

--
-------------------------------------------------------------------------------
--                                  PROCEDURE
--
-- MODULE:  deallocate_c
-- PACKAGED IN:  c_interface
-- SEPARATE OF:
-- VERSION:  1.0
-- WRITTEN:  09/22/92
-- SCCS FILE:
--
-- DESCRIPTION:
--
--      This is a string manipulation routine designed to free the
-- memory for a type 'c_string' variable.  This is done via the UNIX
-- system routine 'free'.
--
-- SEPARATES:
--
-- PACKAGES WITH'D:
--
--      None.
--
-- PACKAGES USE'D:
--
--      None.
--
-- ROUTINES CALLED:
--
--      free
--
-- CALLED BY:
--
-- CALLING SEQUENCE:
--
--   in_str : c_string;
--
--   deallocate_c(in_str);
--
```

```
- MODIFICATIONS:
--       Date          Modifier OS Version                 Modification
--       -------- -    ------- ----------- ------------------------------------------
--
-------------------------------------------------------------------------------------
--
procedure deallocate_c(in_str: c_string) is
  status : integer;
begin
  status := free(c_to_address(in_str));
exception
  when others =>
    my_mesg := "DEALLOCATE_C: memory deallocation error" & ASCII.NUL;
    sm_emsg(my_mesg(1)'address);
end deallocate_c;

end c_interface;
```